Building Web Applications with Maven 2

by Will Iverson
03/01/2007

You may have heard of Maven 2--it's often touted by technologists as a replacement for Ant. You may have even taken some time to browse around on the Maven 2 site, but maybe the documentation has left you a little bit unclear on where and how to go about getting started.

In this article, we will take a look at using Maven 2 to help build a simple web application (a bit of business logic in a JAR and a JSP-based web application). By the end of this article, you should feel comfortable working with Maven 2, and ready to start using it as a much more satisfactory tool than Ant (or even your IDE).

Getting Started

These instructions assume that you have installed Java 5 and Maven 2. The following two commands shown should work at your command line:

```
C:\>java -version
 java version "1.5.0_06"
 Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_06-b05)
 Java HotSpot(TM) Client VM (build 1.5.0_06-b05, mixed mode, sharing)

C:\>mvn -v
Maven version: 2.0.5
```

Everything else required for this project will be downloaded for you automatically by Maven 2 (obviously, a working internet connection is also required). I used my Windows system to write this article, but everything here should work fine on Mac OS X, Linux, Solaris, etc.

From a high level, the project will be organized into two subprojects (one for the JAR and one for the WAR). Let's start by creating the base directory for the project. This directory serves as the base for the other folders.

```
C:\>mkdir maven2example
C:\>cd maven2example
C:\maven2example>
```

Now, let's create the two subprojects. Maven 2 supports the notion of creating a complete project template with a simple command. The project templates (called “archetypes” in Maven) shown below are a subset of the full list of archetypes built in to Maven 2.

Project Template (Archetype)	Purpose
maven-archetype-archetype	Create your own project template (archetype).
maven-archetype-j2ee-simple	Creates a J2EE project (EAR), with directories and subprojects for the EJBs, servlets, etc.
maven-archetype-mojo	Create your own Maven 2 plugins.
maven-archetype-quickstart	Simple Java project, suitable for JAR generation. Maven 2 default.
maven-archetype-site	Documentation-only site, with examples in several formats. You can run this archetype on top of an existing Maven 2 project to add integrated documentation.

maven-archetype-webapp	Creates a web application project (WAR), with a simple Hello World JSP.

These archetypes are analogous to the sample projects you might find in your IDE as defaults for standard "New Project…" options.

To create a simple Java project, you simply execute the command as shown.

```
mvn archetype:create
   -DgroupId=[your project's group id]
   -DartifactId=[your project's artifact id]
```

The `mvn` command invokes the Maven 2 system, in this case a request to run the `archetype` plugin with the `create` command. The `-D` commands are simply setting Java system properties, thereby passing configuration information to Maven 2. Per the Maven 2 FAQ, the `groupId` should follow the package name (reversed DNS of your website), and can contain subgroups as appropriate. For example, the sample code for my books might use `com.cascadetg.hibernate` or `com.cascadetg.macosx`. In this case, let's use the `com.attainware.maven2example`.

The `artifactId` is specific to each artifact and by convention should be the filename, excluding extension. In this case, we would like to create two artifacts, a JAR file containing the logic and a WAR file containing the web application. First, let's create the JAR file using the command as shown below. Maven 2 tends to be quite verbose, and so I have trimmed the output shown here and and many of the other listings in this article to focus on the elements of interest.

```
C:\maven2example>mvn archetype:create
        -DgroupId=com.attainware.maven2example
      -DartifactId=maven2example_logic

[INFO]   Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO]   ------------------------------------------------------
[INFO] Building Maven Default Project
   (snip)
[INFO] Archetype created in dir:
                     C:\maven2example\maven2example_logic
[INFO]   ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO]   ------------------------------------------------------
[INFO] Total time: 1 second
[INFO] Finished at: Sun Feb 04 10:42:33 PST 2007
[INFO] Final Memory: 4M/8M
[INFO]   ------------------------------------------------------

C:\maven2example>
```

Looking at the resulting file structure, we can see that Maven 2 has done several things for us. It's created a fairly complete directory structure, following many best practices for organizing code. The source code is broken into two directories, one for the code itself and one for the test cases. The package structure is taken from the `groupId`, and is mirrored in both the main code and the test case directory structure.

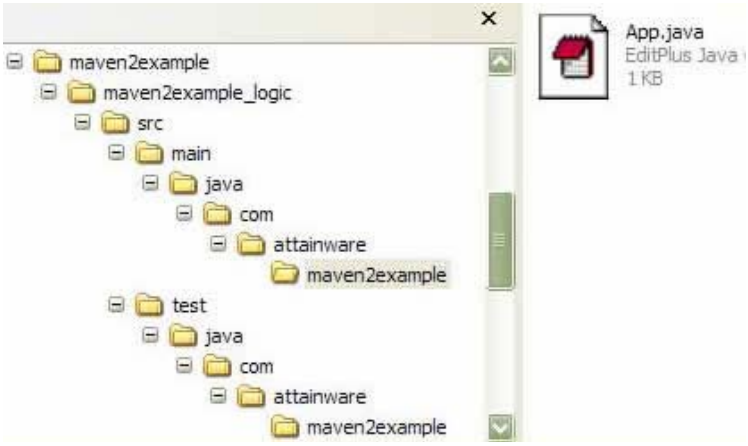

Figure 1. Source for JAR project layout

Clicking through the results, you will see folders and two Java source files--so far, nothing surprising.

In the root folder, however, we notice a file called *pom.xml*.

Figure 2. Maven 2 project file for JAR project

This file is essentially analogous to an IDE project file: it contains all of the information about a project, and is the file that Maven 2 uses to act upon to execute commands. The contents of *pom.xml*:

```
<project xmlns= http://maven.apache.org/POM/4.0.(
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance'
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.attainware.maven2example</groupId>
        <artifactId>maven2example_logic</artifactId>
     <packaging>jar</packaging>
     <version>1.0-SNAPSHOT</version>
     <name>maven2example_logic</name>
     <url>http://maven.apache.org</url>
     <dependencies>
       <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
           <version>3.8.1</version>
         <scope>test</scope>
       </dependency>
     </dependencies>
 </project>
```

There really isn't a lot in the standard *pom.xml* file. We can see the `groupIc` and `artifactIc`, that the project is intended to build a JAR, and that the current version is to 1.0-SNAPSHOT. The project uses JUnit 3.8.1 for unit tests. We could change the URL to point to our company or project website and/or update the version number to something more appropriate, but for now this is fine. If you are curious, you can review the schema for Maven *pom.xml* files.

Maven 2 makes heavy use of standard directory layouts to reduce clutter. It assumes that the source code and test code will be found in the directory created by the project template. By specifying standard directory layouts, it easier to immediately start work on a project with spending a lot of time relearning the build process. While it is possible to reconfigure Maven 2 to use custom directory layouts, I have found that it's generally less work to simply use the Maven 2 layout. I have converted several projects of small to medium size from Ant to Maven 2, and found that the size and complexity of my build files (from custom Ant *build.xml* files to Maven 2 *pom.xml* files) dropped by an order of magnitude. By using Maven 2 templates (archetypes) and simply copying source files into the proper locations, I found that I wound up with much more comprehensible project structures than the slowly accreted, custom Ant *build.xml* projects. While it is possible to force Maven 2 to fit into arbitrary directory structures, that's probably not the place to start.

Maven 2 Commands

Maven 2 supports two kinds of commands that can be run on projects (*pom.xml* files). The first type of command is a *plugin* command. Plugin commands include things like "copy a set of files," "compile a source tree," etc. The other type of command is a *lifecycle* command. A lifecycle command is a series of plugin commands strung together. For example, the test lifecycle command might include several plugin commands in a series.

Let's execute the lifecycle command `mvr test` on our *pom.xml* file. As can be seen in the following listing, this command executes several plugins (additional output omitted for readability).

```
C:\maven2example\maven2example_logic>mvn test

[INFO] Scanning for projects...
[INFO]   ------------------------------------------------------
[INFO] Building maven2example_logic
[INFO]    task-segment: [test]
[INFO]   ------------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
[INFO] [surefire:test]
[INFO]   ------------------------------------------------------
[INFO] BUILD SUCCESSFUL
```

```
C:\maven2example\maven2example_logic>
```

What we can see is that the `mvn test` lifecycle command is bound to several plugins, including `resources`, `compiler`, and `surefire`. These plugins in turn are called with different goals, such as `compile` or `testCompile`. So, by simply calling `mvn` with a single lifecycle command, we execute a number of plugins--no additional configuration necessary. If you want to specify a plugin directly, that's fine. For example:

```
C:\maven2example\maven2example_logic>mvn compiler:compile

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'compiler'.
[INFO] ------------------------------------------------------------
[INFO] Building maven2example_logic
[INFO]    task-segment: [compiler:compile]
[INFO] ------------------------------------------------------------
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------

C:\maven2example\maven2example_logic>
```

Now that we have seen both lifecycle commands and plugin commands in action, let's look at some of the typical lifecycle commands.

mvn clean	Cleans out all Maven-2-generated files.
mvn compile	Compiles Java sources.
mvn test-compile	Compiles JUnit test classes.
mvn test	Runs all JUnit tests in the project.
mvn package	Builds the JAR or WAR file for the project.
mvn install	Installs the JAR or WAR file in the local Maven repository (use this if you have multiple interdependent local projects).

A complete list of lifecycle commands can be found at on the Maven 2 site.

Similarly, here is a list of some of the more popular plugin commands, analogous to the popular tasks in Ant or commands in an IDE. Note that a lifecycle command name is a single word, whereas a plugin command name is broken into a plugin and a specific goal with a colon:

clean:clean	Cleans up after the build.
compiler:compile	Compiles Java sources.
surefire:test	Runs the JUnit tests in an isolated classloader.
jar:jar	Creates a JAR file.
eclipse:eclipse	Generates an Eclipse project file from the *pom.xml* file.

You may want to take some time perusing the list of plugins that are available to Maven 2 by default. It is possible to add additional plugins by either writing them yourself or by grabbing them from a repository (repositories are discussed later in this article).

You can configure plugins on a per-project basis by updating your *pom.xml* file. For example, if you wish to force compilation on Java 5, simply pass in the following option under the build configuration in the *pom.xml* file:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi=  "http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=  "http://maven.apache.org/POM/4.0.0
```

```
http://maven.apache.org/maven-v4_0_0.xsd">
       <build>
         <plugins>
           <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
              <configuration>
                   <source>1.5</source>
                   <target>1.5</target>
              </configuration>
           </plugin>
         </plugins>
       </build>
 </project>
```

Each plugin listed on the Maven 2 site includes documentation on the various configuration options available.

Maven 2 Versus Ant?

If you are familiar with Ant, you may be wondering why lifecycles and plugins are an improvement over targets and tasks. While you can cleanly separate your Ant targets from the various directories and artifacts, it's surprisingly difficult to do. For example, let's say that you have several directories, with several artifacts (multiple JAR files, which get combined into a WAR, for example). Every command that you write for each Ant target will require properties for the source, and various dependencies, typically passed around via a combination of Ant variables. Dependencies in particular require a lot of custom handholding, with JAR files downloaded and managed by hand.

Maven 2, on the other hand, assumes that given a *pom.xml* file, you are probably always going to be doing the same general kind of things with your code. You are probably going to want to build a JAR file from that lump of code over *there*. You are probably going to want to build a WAR file from that set of files over *there*. So those actions (or goals) are written in Java as reusable chunks of code (plugins). Instead of complicated lumps of thousands of lines of custom *build.xml* files, you just call a plugin with a specific goal.

There are several advantages to this approach. For one, the Maven 2 Java-based plugins are a lot smarter than the lower-level commands found in Ant tasks. You don't have to write custom tasks that handle compilation, and then have to worry about keeping your clean task in sync with your compile and packaging tasks; Maven 2 just takes care of this stuff for you.

Another advantage of this model is that it encourages reuse of plugins in a much more robust way than Ant tasks. Later in this article, we will show how simply adding a few lines to a *pom.xml* file will automatically download and launch Jetty (a lightweight servlet/JSP container), seamlessly installing and running the WAR file.

Adding Some Business Logic

Let's add a tiny bit of pseudo-business logic to our application. First, we write a simple test cast that looks for a String, updating *C:\maven2example\maven2example_logic\src\test\java\com\attainware\maven2example\AppTest.java*.

```
package com.attainware.maven2example;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import com.attainware.maven2example.App;

public class AppTest extends TestCase
{
       public void testApp()
       {
           assertTrue( App.now().length() > 0 );
       }
}
```

Opening the file *C:\maven2example\maven2example_logic\src\main\java\com\attainware\maven2example\App.java*, we update the contents to:

```
package com.attainware.maven2example;

public class App
{
       public static String now()
       {
            return new java.util.Date().toString();
       }
}
```

Running the command `mvn install`, as shown below, causes several plugins to run (status messages omitted for readability).

```
C:\maven2example\maven2example_logic>mvn install

[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
[INFO] [surefire:test]
[INFO] [jar:jar]
[INFO] [install:install]
[INFO] BUILD SUCCESSFUL

C:\maven2example\maven2example_logic>
```

By running the `mvn install` lifecycle command, Maven 2 finishes by installing the JAR file into the local repository, making this resulting JAR file available to other projects on this system. This allows this JAR file to be available for inclusion in our WAR file.

Creating the WAR file

Now that we have created our JAR file, creating the WAR file is straightforward--we just pass in a different archetype ID.

```
C:\maven2example>mvn archetype:create
    -DgroupId=com.attainware.maven2example
    -DartifactId=maven2example_webapp
    -DarchetypeArtifactId=maven-archetype-webapp

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] ------------------------------------------------------------
[INFO] Building Maven Default Project
[INFO] [archetype:create]
[INFO] Archetype created in dir:
                    C:\maven2example\maven2example_webapp
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------

C:\maven2example>
```

As shown in Figure 3, we now have a directory structure you would expect to see for a WAR. Note that the JAR file is automatically pulled into the WAR file from the local repository.

Figure 3. WAR project directory structure

Simply executing the command `mvn package` creates our WAR file (again, omitting most of the status messages besides the plugin list).

In this case, the `mvn package` command builds the WAR file but does not install it into the local repository (repositories will be discussed later). As you build larger, more sophisticated projects with more complex dependencies, you may wish to be more deliberate about using the `mvn package` and `mvn install` commands to control inter-project dependencies. Because the WAR file is the final artifact we are creating, we can save a few compute cycles merely by packaging the WAR file in the local target directory.

```
C:\maven2example\maven2example_webapp>mvn package

[INFO] Scanning for projects...
[INFO] ------------------------------------------------------------
[INFO] Building maven2example_webapp Maven Webapp
[INFO]    task-segment: [package]
[INFO] ------------------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
[INFO] [surefire:test]
[INFO] [war:war]
[INFO] Building war:
        C:\maven2example\maven2example_webapp\
            target\maven2example_webapp.war
[INFO] ------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ------------------------------------------------------------

C:\maven2example\maven2example_webapp>
```

As we can see in Figure 4, we now have a WAR file.

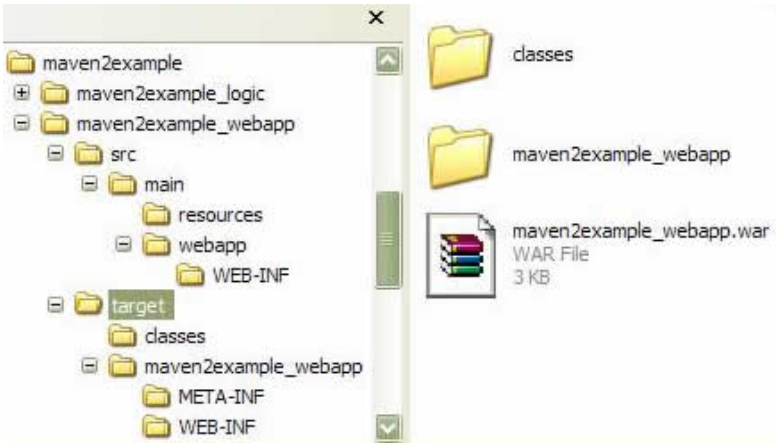

Figure 4. Resulting WAR file

Now that we have a WAR file, we would like to be able to run the application. To do this, we simply add the Jetty plugin to the *pom.xml* for this web application as shown:

```
<project xmlns=  "http://maven.apache.org/POM/4.0.0'
   xmlns:xsi=  "http://www.w3.org/2001/XMLSchema-instance'
  xsi:schemaLocation=  "http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.attainware.maven2example</groupId>
     <artifactId>maven2example_webapp</artifactId>
     <packaging>war</packaging>
     <version>1.0-SNAPSHOT</version>
     <name>maven2example_webapp   Maven Webapp</name>
     <url>http://maven.apache.org</url>
     <dependencies>
       <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
           <version>3.8.1</version>
         <scope>test</scope>
       </dependency>
     </dependencies>
     <build>
       <finalName>maven2example_webapp</finalName>
         <plugins>
              <plugin>
                   <groupId>org.mortbay.jetty</groupId>
                   <artifactId>maven-jetty-plugin</artifactId>
              </plugin>
          </plugins>
   </build>
 </project>
```

By adding the plugins entry with the Jetty information, we can now run our web application by simply typing the command as shown below.

```
C:\maven2example\maven2example_webapp>mvn jetty:run

[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO]   ------------------------------------------------------
[INFO] Building maven2example_webapp Maven Webapp
[INFO]    task-segment:   [jetty:run]
[INFO]   ------------------------------------------------------
[INFO] Preparing jetty:run
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [jetty:run]
[INFO] Starting jetty 6.1.0pre0   ...
[INFO] Classpath =
  [file:/C:/maven2example/maven2example_webapp/target/classes/]
  2007-02-04 12:41:24.015::INFO:
   Started SelectChannelConnector @   0.0.0.0:8080
[INFO] Started Jetty Server
```

We can now open our web browser to localhost:8080/maven2example_webapp to see our web application (as shown in Figure 5). Type `Ctrl-C` in the console window to shut down the server. Note that we didn't have to download and install Jetty separately--Maven 2 automatically downloads and configures Jetty.

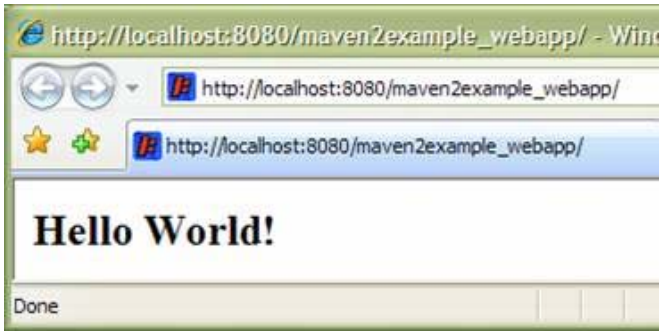

Figure 5. Hello World Web Application

Tying Together the Logic and Web Application

Now, we simply have to tie together the logic in our JAR file and the web application. First, we want to set up a dependency between our JAR and the web application. This will tell Maven 2 that we want to use this JAR file in our WAR, which will cause Maven 2 to automatically copy the JAR file when we package our WAR file.

First, we add the dependency on this other JAR file to our *pom.xml* for the web application as shown below.

```
<project xmlns=  "http://maven.apache.org/POM/4.0.0'
 xmlns:xsi=  "http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation=  "http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.attainware.maven2example</groupId>
     <artifactId>maven2example_webapp</artifactId>
     <packaging>war</packaging>
     <version>1.0-SNAPSHOT</version>
     <name>maven2example_webapp   Maven Webapp</name>
     <url>http://maven.apache.org</url>
     <dependencies>
       <dependency>
         <groupId>junit</groupId>
         <artifactId>junit</artifactId>
           <version>3.8.1</version>
         <scope>test</scope>
       </dependency>
       <dependency>
            <groupId>com.attainware.maven2example</groupId>
             <artifactId>maven2example_logic</artifactId>
            <version>1.0-SNAPSHOT</version>
       </dependency>
     </dependencies>
     <build>
       <finalName>maven2example_webapp</finalName>
         <plugins>
              <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>maven-jetty-plugin</artifactId>
              </plugin>
         </plugins>
     </build>
 </project>
```

We then update our JSP file to use our fancy new business logic.

```
<html>
 <body>
 <h2>Fancy Clock</h2>
 <%= com.attainware.maven2example.App.now() %>
 </body>
</html>
```

Next, we run the command `mvn package` to rebuild the WAR file.

```
 C:\maven2example\maven2example_webapp>mvn package

 [INFO] Scanning for projects...
 [INFO]   ------------------------------------------------
 [INFO] Building maven2example_webapp Maven Webapp
 [INFO]    task-segment:   [package]
 [INFO]   ------------------------------------------------
 [INFO] [resources:resources]
 [INFO] [compiler:compile]
 [INFO] [resources:testResources]
 [INFO] [compiler:testCompile]
 [INFO] [surefire:test]
 [INFO] [war:war]
 [INFO] Building war:
     C:\maven2example\maven2example_webapp\
          target\maven2example_webapp.war
 [INFO]   ------------------------------------------------
```

```
[INFO] BUILD SUCCESSFUI
[INFO] ------------------------------------------------------------
```

Notice that the JAR file is now copied into WAR file, as shown in Figure 6.

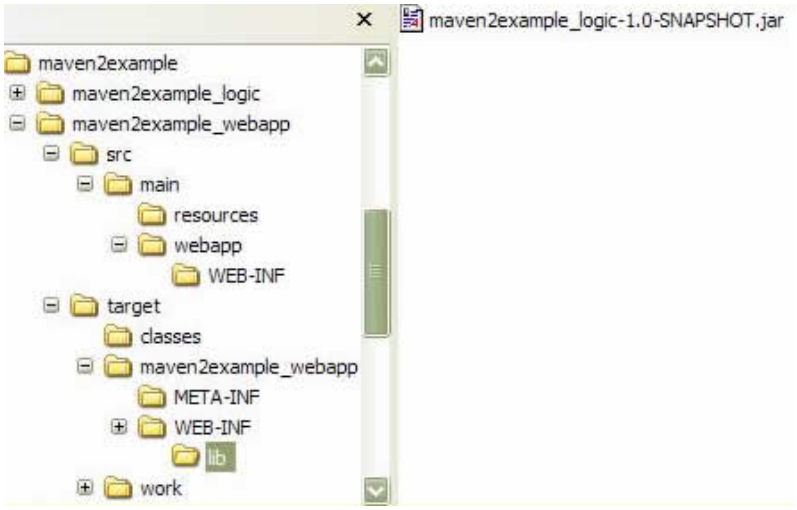

Figure 6. Verifying the JAR copied into the WAR

When we execute the `mvn jetty:rur` command and view the results in our browser, we see the results as shown in Figure 7.

Figure 7. WAR running with business logic from JAR

To recap, we now have two independent projects running on our system. When we run `mvn install` on the logic, the JAR file installed in the local repository will be updated. When we run `mvn package` on the web application, it will pick up the latest copy installed into the local repository.

We would like to be able to run a single command to update both the JAR file and the WAR file. To do this, we create another *pom.xml* file that invokes both projects.

```
<project>
     <name>Maven 2 Example</name>
         <url>http://www.attainware.com/</url>
     <modelVersion>4.0.0</modelVersion>

     <groupId>com.attainware.maven2example</groupId>
        <version>1.0</version>
     <artifactId>maven2example_package</artifactId>
     <packaging>pom</packaging>

     <modules>
       <module>maven2example_logic</module>
       <module>maven2example_webapp</module>
     </modules>
 </project>
```

This *pom.xml* file lives right above the other projects and serves as a "master" project file.

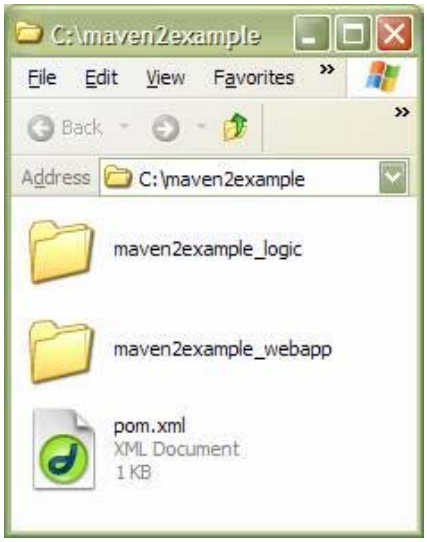

Figure 8. Location of master project file

Now, we can simply execute a single command, `mvn install`, and do a full build on both projects.

```
C:\maven2example>mvn install

[INFO] Scanning for   projects...
[INFO] Reactor build order:
[INFO]   maven2example_logic
[INFO]   maven2example_webapp Maven Webapp
[INFO]   Maven 2   Example
[INFO]   ----------------------------------------------------
[INFO] Building maven2example_logic
[INFO]    task-segment: [install]
[INFO]   ----------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
[INFO] [surefire:test]
[INFO] [jar:jar]
[INFO] [install:install]
[INFO]   ----------------------------------------------------
[INFO] Building maven2example_webapp Maven Webapp
[INFO]    task-segment:   [install]
[INFO]   ----------------------------------------------------
[INFO] [resources:resources]
[INFO] [compiler:compile]
[INFO] [resources:testResources]
[INFO] [compiler:testCompile]
[INFO] [surefire:test]
[INFO] [war:war]
[INFO] [install:install]
[INFO] [site:attach-descriptor]
[INFO] [install:install]
[INFO]   ----------------------------------------------------
[INFO] Reactor Summary:
[INFO]   ----------------------------------------------------
[INFO] maven2example_logic   ................ SUCCESS [2.281s]
[INFO] maven2example_webapp Maven Webapp .... SUCCESS [0.563s]
[INFO] Maven 2 Example   ................... SUCCESS [1.156s]
[INFO]   ----------------------------------------------------
[INFO] BUILD SUCCESSFUL
[INFO] ----------------------------------------------------
```

To summarize, we now have a project that now compiles and tests a JAR, which is then built and installed into a WAR, which is then in turn installed into a local web server (which is downloaded and automatically configured). All of this with a few tiny *pom.xml* files.

Repositories

You may be wondering about the use of the term "repository." We have glossed over the use of the term, but in brief, Maven 2 makes use of two kinds of repository: local and remote. These repositories serve as locations for Maven 2 to automatically pull dependencies. For example, our *pom.xml* file above makes use the local repository for managing the dependency on the JAR file, and the default Maven 2 remote repository for managing the dependencies on JUnit and Jetty.

Generally speaking, dependencies come from either the local repository or remote repositories. The local repository is used by Maven 2 to store downloaded artifacts from other repositories. The default location is based on your system. Figure 9 shows the local repository on my laptop as of the writing of this article.

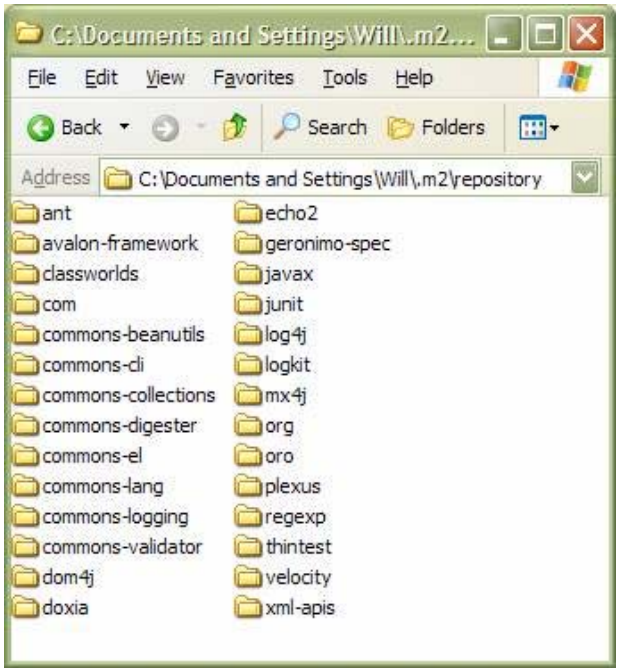

Figure 9. Local repository example

If Maven 2 can't resolve a dependency in the local repository, it will try to resolve the dependency using a remote repository.

The default remote repository, known as Ibiblio, includes many of the most popular open source packages. You can browse the wide range of packages on Ibiblio with the URL above and add dependencies as needed. For example, let's say that you would like to use Hibernate in your project. Navigating to www.ibiblio.org/maven/org.hibernate/poms, we can see there are a wide number of releases of Hibernate available. Opening up a sample Hibernate *pom* file, we can see that we simply need to add the appropriate `groupIc`, `artifactIc`, and `versior` entries to our business logic *pom.xml* file (the scope flag tells Maven 2 which lifecycle is interested in the dependency).

```
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate</artifactId>
        <version>3.2.1.ga</version>
        <scope>compile</scope>
 </dependency>
```

Simply adding this dependency to the *pom.xml* file will cause Maven 2 to automatically download Hibernate and make the appropriate JAR files available to both the business logic JAR file and the WAR file.

You can set up your own remote repository, and add an entry to the *pom.xml* file to look in that repository for artifacts. This is extremely useful for enterprises that make use of shared resources (for example, one group may wish to publish JAR files that are used to access a particular piece of middleware).

Finally, you may instead wish to install your own JARs into the local repository. For example, if you have a *Simple.jar* file that someone gave you, use the command shown below (choosing `groupIc` and `artifactIc` values that are highly likely to be unique to avoid a namespace collision).

```
mvn install:install-file
     -Dfile=Sample.jar
     -DgroupId=uniquesample
     -DartifactId=sample_jar
     -Dversion=2.1.3b2
     -Dpackaging=jar
     -DgeneratePom=true
```

Summary

In this article, we looked at how a few commands and some tiny XML files allow us to create, compile, test, bundle, and manage project dependencies. We built a simple web application and deployed it to a web server with just a few commands, and we still haven't touched on many of the features of Maven 2. For example, additional commands generate integrated Javadocs across multiple projects, code coverage reports, or even a complete website with documentation. With luck, this orientation to Maven 2 has given you enough information to begin the transition. Eventually, tools such as Eclipse and NetBeans will almost certainly support Maven 2 (or something like it) natively. In the meantime, you can dramatically reduce your use of raw Ant (and spend a lot less time fighting XML build scripts) by switching even small projects over to Maven 2.

Resources

- Source code shown in this article is available for download from www.cascadetg.com/maven.
- The main Maven 2 site

Will Iverson is the practice director, Software Development for SolutionsIQ, a Pacific Northwest service provider.

java.net RSS Feeds